SuperGlue: A Programming Environment for Scientific Visualization

J.P.M. Hultquist
hultquist@nas.nasa.gov

E.L. Raible
raible@nas.nasa.gov

July 1992
Report RNR-92-014

Numerical Aerodynamic Simulation Systems Division
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, CA 94035-1000

```
@inproceedings{xxx,
 author    = "J.P.M. Hultquist and E.L. Raible",
 title     = "SuperGlue: A Programming Environment
              for Scientific Visualization",
 booktitle = "Proceedings of Visualization '92",
 address   = "Boston, MA",
 month     = oct,
 year      = 1992,
 pages     = "243-250",
 note      = "(also RNR Tech Report 92-014)"}
```

Abstract

Visualization environments have two audiences: scientists and programmers. We suggest that many existing platforms overemphasize ease-of-use and do not adequately address issues of extensibility. We have built a visualization testbed, called SuperGlue, which is particularly suited for the rapid development of new visualization methods. An interpreter supports rapid development of new code and an extensive class hierarchy encourages code reuse.

By explicitly designing for ease of programming, we have produced a visualization system which is powerful, easy to use, and rapidly improving. This report describes the motivation of the work, the architecture of the system, and our plans for further development.

1 Introduction

Our application is the visualization of computational fluid dynamics (CFD) simulation results. This is a particularly demanding field due to the complex nature of the datasets. A steady-state 3D solution is produced by solving the Navier-Stokes equations (a system of non-linear partial differential equations) in a curvilinear coordinate system. This calculation typically takes many hours of Cray-class supercomputer time. The resulting dataset consists of a collection of vector and scalar fields sampled at millions of points. To meet the challenge of visualizing such data, we have produced an environment explicitly tailored for the rapid development of new visualization techniques for CFD.

1.1 Requirements

Scientific visualization packages share a common set of requirements. Although most packages attempt to meet them, most fall short in several ways. These requirements are not unique to visualization, but they apply more strongly to it than to many other specialties.

extensibility Cooperation between scientists and programmers can be very fruitful, but the value of this relationship depends upon the ability of the programmers to rapidly implement new features in response to changing needs.

flexibility Scientific visualization software must implement a wide variety of requirements including: reacting to user inputs, displaying computed models, and performing complex application-specific calculations over very large datasets.

robustness Visualization in many disciplines requires the management of a large amount of data, examined using many techniques. The development of new visualization methods is best achieved in the context of a running environment, and errors in the new code should not be allowed to bring down a running environment and its built-up internal state.

1.2 Existing platforms

In this section, we consider three popular systems for the visualization of CFD results: Plot3D [4], FAST [1], and Explorer [9] from Silicon Graphics.

Plot3D is a mature, straightforward, and highly portable system. It is organized as a single-process program with a command-line interface. It has enjoyed wide acceptance because its pragmatic approach solves many problems that CFD researchers face on a daily basis.

In the FAST model, separate processes communicate through shared memory to produce a collection of geometric models of the data. These models are displayed by a central rendering module. New application-specific tools can attach to the shared memory, and may place newly-computed data and models into this common pool.

Explorer (and its conceptual predecessor AVS [13]) are examples of the "data flow paradigm" of scientific visualization. New modules or *filters* can be strung together in a network through which data is processed from input file to screen image. A graphical network editor provides a simple means to construct a wide variety of different visualizations.

1.3 Limitations

All of these systems are extensible to some degree, but none of them strongly addresses the issue of rapid prototyping, full extensibility, or code reuse. Although FAST and Explorer allow the recompilation of just the single module under development, even this requirement can be disruptive to the programmer. Plot3D requires full recompilation in order to test new extensions.

Both FAST and Explorer offer some library-based utility routines for the implementation of new modules, but otherwise provide no provisions for reusing code. Plot3D provides no explicit support for code reuse, but has been extensively customized in dozens of derivative versions.

1.4 SuperGlue

It should be clear that the previously discussed requirements of a visualization system (extensibility, flexibility, and robustness) are best designed into a system from the start; it is extremely difficult to add them after the fact.

Of these three requirements we assert that *system-wide* extensibility is the most difficult to achieve. One common approach is to include a specialized configuration language. It has been our experience that these *ad hoc* languages grow over time and rarely develop into a coherent whole.

Rather than modify a visualization system to meet our requirements, we started with a system which already met these requirements and then added the specific features needed for visualization. More specifically, we centered the system around a complete, interpreted language – a dialect of Lisp called Scheme [11] – which we then extended to support the demands of visualizing large CFD datasets.

SuperGlue has been implemented on SGI hardware; it should be portable, however, to any machine which supports a rich graphics environment.

1.5 Previous extension languages

The extension language approach has been used in many applications.

AutoCAD is a widely-used commercial system which uses a Lisp-like language for customization and extension. GNU Emacs also incorporates an interpreter for a dialect of Lisp, combined with a large number of compiled primitive functions for processing text. Beckman [2] used Scheme to implement a system for interpreted programming with the Silicon Graphics GL.

P3D (Welling *et al* [14]) is a restricted dialect of Common Lisp used to describe graphical models of the scientific data. SuperGlue extends this notion by allowing the language to describe user interaction as well as the format and the graphical presentation of the data.

PDBq (Palmer [10]) is a scientific visualization package which is built upon an special-purpose language which resembles C. Explicit support for vector arithmetic simplifies the programming of new visualization methods.

1.6 Overview

Section 2 describes the overall architecture of SuperGlue. Sections 3 and 4 discuss the major extensions which were added to the command language. The application and the programming interfaces of SuperGlue are described in section 5, and an example implementation of a new feature is examined in section 6. The paper concludes with a review of the benefits and the difficulties of this approach, and a brief discussion of our future plans.

2 A dual-language architecture

The SuperGlue environment has been implemented using two languages: Scheme and C. This split adds some conceptual burden, but offers many advantages which make this approach worthwhile.

2.1 Scheme as a control language

Interactive visualization requires a flexible and intuitive user interface, which is difficult to develop out of thin air. It must be evolved with the benefit of experience and experimentation. Scheme has several characteristics that make it the ideal choice for implementing the flow of control of complex interfaces:

- It is *interpreted*. Programs can be written and tested interactively.
- It is *complete*. Scheme is able to express a wide variety of programming paradigms.
- It is *standardized*. This allows us to share code with other researchers.
- It is *convenient*. Scheme provides dynamic typing, automatic memory deallocation, simple syntax, and a wide variety of data types.

SuperGlue is based on the "Xscheme" program written by David Betz, who allows its use for non-commercial purposes. Xscheme compiles Scheme expressions into machine-independent byte-codes, which are then executed on a virtual machine implemented in C. The result is a reasonable compromise between portability, compilation time, and execution speed.

2.2 C as a computation language

Although Scheme is a fine language for high level control and user interface programming, most of our numerical and graphical primitives are coded in C. Execution speed is the primary reason for this, though many programmers also prefer C's compact syntax for mathematical programming. C is also more convenient when interfacing to Unix libraries. (Nothing in principle prevents new primitives from begin written in C++, FORTRAN, or other compiled languages, but at present it is less convenient.)

Another reason to use C is multiprocessing. Our SGI workstations support the creation of autonomous lightweight threads that run on multiple CPUs. Once instantiated, a thread will run asynchronously until it needs to return a result. It must then gain temporary exclusive access to the interpreter in order to avoid corrupting shared data structures. A set of C macros simplifies this synchronization.

In order to maintain the benefits of an interpreted language when programming in C, we have extended Xscheme to support "dynamic loading." This feature allows new versions of a C function to be compiled and loaded into the running system; thereby short-circuiting the traditional edit, compile, link, run, configure, and test cycle. Dynamically loading a page or two of source code takes only five to ten seconds.

2.3 The inter-language interface

A convenient interface between the two languages is a crucial component of any dual-language architecture. Two issues must be addressed: different storage models and different calling conventions.

The difference in storage models is inherent in the specifications of the two languages. In C, *variables* are typed and *values* are untyped; in Scheme, *values* are typed and can be assigned to any variable. (This is equivalent to saying the C does compile-time type checking, while Scheme does run-time type checking.) We were able to preserve Scheme's "exceptionally clear and simple semantics"[11] by making it the responsibility of every C primitive to type-check and convert each of its Scheme arguments into the corresponding C format. When the function completes, its result must be converted to a Scheme data item.

The issue of different calling conventions is a consequence of the Xscheme virtual machine architecture. This virtual machine manages its own stack, which is independent of the C hardware stack. As a result, C primitives must explicitly pop their arguments from the Xscheme stack when they are called, and must explicitly push any arguments onto this stack when they call back to Scheme.

We decided to insulate all Scheme code from these implementation considerations. A set of C macros and functions allows these incompatibilities to be resolved in a straightforward and portable manner. The result is transparent access from Scheme to C and convenient access from C to Scheme. The cost of these conversions in terms of coding, maintenance, and execution time is insignificant.

3 The object system

The advantages of object-oriented systems are numerous and well-documented. The Xscheme kernel has extended the Scheme language to support single inheritance with dynamic method lookup. We added a convenient syntax for defining new methods and have built a class hierarchy of over 150 classes and almost 1500 methods. This class hierarchy collectively defines the generic features of all interactive visualization applications.

3.1 The hierarchy

The class `<object>` defines the behavior of all instances in the system, including that of the class `<class>`. The class `<class>` defines the behavior of all classes in the system, including itself. The remaining classes are grouped into the following major subtrees:

`data` contains the essential "computer science" data structures such as collections, stacks, queues, and dictionaries.

`graphics` implements an object-oriented interface to the Silicon Graphics GL. Automatic deallocation of GL resources is supported. Higher-level graphical objects, such as curves and meshes, are also included. Finally, direct-manipulation is supported by a virtual trackball and a 3D cursor.

`interface` provides 2D user-interface objects (such as sliders, buttons, and popup menus) implemented using GL. A *scene* widget forms a rectangular picture of a 3D scene.

`math` provides access to application-specific compiled code, such as numerical integration and interpolation routines. Class definitions are provided for matrices, quaternions, and the composite structures which represent CFD data sets.

`system` includes interfaces to Unix resources such as files, directories, and threads. Additional classes implement the Scheme runtime environment; these include clocks, alarms, and a source-level Scheme debugger.

3.2 Defining a new class

Each class inherits some of its behavior and contents from its superclass. It adds to this initial configuration by adding additional internal state, and by defining new methods which can be used to modify that state.

```
(defclass class-name
    (super superclass-name)
    (class class-variable
          ( class-variable initializer)
           ...)
    (instance instance-variable
              ...))
```

The `defclass` statement begins with the name of the new class, followed by three statements which define the superclass and the additional internal state of the new class. The `class` statement defines the variables which are shared among all instances of the new class. The `instance` statement defines the new variables which are held as separate copies in each new instance.

```
(defmethod class-name ( method-name arg1 ...)
     body ...))
```

The `defmethod` statement declares the name of the relevant class, the name of the method itself, an argument list, and the body of the method.

3.3 Message sending

The first item of any Scheme expression must be a function or an object. If it is an object, then second item must be a symbol, which is then used as the *message* to be sent to that instance.

```
(define baz (stack 'new))
(baz 'push (+ 2 3))
(baz 'push (* 3 4))
(baz 'pop)      → 12
(baz 'empty?)     → false
```

Messages which cannot be handled by the immediate class of the receiving instance are passed up the superclass chain. If a message fails to match any method in the chain, the erroneous send is trapped in the source code debugger.

4 Storage management

A typical CFD dataset consists of a grid of a million points with a 3D position and three sampled flow quantities recorded for each point. As scientists examine a solution, they typically calculate many more fields over that same grid. Several features were explicitly added to SuperGlue to support the management of these large datasets.

4.1 Chunks

Chunks are Scheme data items which contain a header and a contiguous block of untyped memory. The information in the header allows the garbage collector to reclaim storage when appropriate. Compiled primitives are used to process the compact data within the chunk.

Some scientific datasets are extremely large; so large that if we were to read them with normal methods we would rapidly exhaust the available swap space of the average workstation. This fatal situation can be avoided by *memory mapping* the contents of these large files directly into the program's address space. *Memory mapped chunks* are similar to normal chunks, except that they rely on the operating system to read the data from disk only when it is referenced. Besides preserving swap space, memory mapped chunks eliminate the initial file-reading delay of most other systems. This is significantly more efficient if only a subset of the data is used.

4.2 Storage of field data

It is often convenient to encapsulate a chunk within an instance of a class. This arrangement gives us the best of both worlds: a Scheme object with a set of supported methods, and an efficient means of storing the data. Taken together, these provide ease of bookkeeping with the speed required for complicated application-specific data manipulation.

CFD simulation results are represented by an instance of the class `<bundle>` (*cf.* Butler [5]), which contains a collection of named fields defined on a common curvilinear grid. Each field contains its sampled volume as a chunk. This chunk is passed to application-specific primitives which operate on fields.

A bundle may be a subvolume of another bundle. When a bundle is asked for a field it does not have, the child may compute that field from the fields it does contain or may extract the data from the corresponding field in its parent. The expressive power of the interpreted language could be used to implement additional bookkeeping to support remote or lazy evaluation of field data.

4.3 Destructors

Certain objects used in SuperGlue have an existence external to the system. For example, the SGI Graphics Library maintains lights, textures, and materials within its own address space. A mechanism for explicit *destructors* has been implemented so that when the associated instance for such an item is reclaimed by SuperGlue, the corresponding external resource can be explicitly deallocated.

The interpreter was extended to maintain a list of items for which explicit "destructor functions" have been defined. When the garbage collector completes its mark pass of memory, unmarked objects on this special list are transferred to a separate list of "reclaimed" objects. Items on this "reclaimed" list share the property that they are no longer in use, and thus can be deallocated when the system becomes idle. This approach consumes only three words of overhead for those few objects which require explicit destruction. It requires only a single scan of the special list between the mark and the sweep phases of the garbage collection.

5 Interacting with SuperGlue

SuperGlue offers two interfaces. One is a mouse-directed interface of widgets and and 3D scenes used in the target application. The other is a textual interface which is of use primarily to programmers.

5.1 The application interface

The point-and-click side of SuperGlue presents a number of *scenes* and *panels*. A scene depicts a set of objects, or *visuals*, in a 3D environment. A scene can be resized to any size or shape, and can even cover the entire workstation screen. The constructed models can be displayed in any number of scenes; each with its own viewing direction and magnification. A panel contains a number of 2D widgets, is usually fairly small, and cannot be resized. Figure 1 shows a typical screen display, including two views of some 3D data (Rogers [12]), an *instance inspector*, and a *hierarchy browser*.

The application interface of SuperGlue provides direct-manipulation of cursors in the 3D volume of a flow field. Chording of the mouse buttons is used to control a modal mapping between the 2D motions of the mouse and the 3D motions of the cursor. Users often manipulate these 3D probes at high magnification while viewing the resultant models from another angle in a second window. This approach allows the precise and direct placement of control points, a vast improvement over what is possible by indirectly controlling these positions using multiple 2D widgets.

The system attempts to maintain a high frame rate by downgrading the quality of the image while it is moving in response to user input. When the user pauses between commands, the system does *not* pause, but uses the otherwise idle machine cycles to improve the accuracy of the presented results.

5.2 The textual interface

SuperGlue programming is qualitatively different than programming in compiled languages. The primary reason for this is the rapid feedback provided by an interpreted language. Rather than typing at the SuperGlue interpreter directly, we rely on the built-in capability of GNU Emacs to control external processes. Thus all text input to and output from SuperGlue goes through Emacs. The Emacs window can be split into sections, which can independently display files of source code, the interpreter transcript, and interactions with the Scheme and the compiled code debuggers.

A typical SuperGlue session lasts somewhere between several hours and several days. During this time many Scheme and C functions are defined and redefined. Emacs supports this activity by allowing either individual functions or entire files of either C or Scheme code to be downloaded to the interpreter.

5.3 Debugging

SuperGlue provides the usual tools of a Lisp-based environment, including code tracing, pretty-printers, break loops, and a simple programmatic interface for handling errors. Unhandled errors and user-generated interrupts cause the system to trap into the interactive debugger. Unlike most C or FORTRAN environments, it is unnecessary to run “under the debugger,” since it is built into the system.

The Scheme debugger allows the user to print the current source code expression, to display the call stack, to examine and modify local variables, to evaluate arbitrary expressions in the context of a particular frame, to set breakpoints, to automatically display the relevant source code in Emacs, and finally to fix and proceed from most errors. The debugger includes online help and offers different modes for beginning and advanced SuperGlue programmers.

The GNU debugger `gdb` is used to debug all C code, including any dynamically-loaded C. No additional effort is required to debug dynamically-loaded code; the required additional bookkeeping is handled automatically when the programmer loads the C code with the standard SuperGlue Emacs keystroke.

6 A programming example

Programming in SuperGlue often consists of writing new methods to improve the user interface or creating a new class to access some new application-specific routine. We describe an example of the latter in this section, then demonstrate the advantages of incorporating new features as first-class entities of an evolving system.

6.1 Streamline calculation

A common technique in flow visualization uses tangent curves through a vector field. When the field is the velocity of the steady fluid, then each curve defines the path traveled by a massless particle drifting through the flow; such paths are called *streamlines*. In this section, we describe an implementation of streamlines for CFD datasets.

A streamline is the solution of the initial value problem posed by a vector field $\vec{u}$ and an initial point $\vec{x}_0$. We need to compute a sequence of points $(\vec{x}_0, \vec{x}_1, \ldots \vec{x}_n)$ such that

$$\vec{x}_{i+1} = \vec{x}_i + \int_{t_i}^{t_{i+1}} \vec{u}(\vec{x}(s))ds$$

for a closely-spaced sequence of values $(t_i \ \forall \ i \in [0, n])$.

This integration requires numerical methods iterated over a piecewise interpolation of the vector field samples. SuperGlue provides several primitives for this calculation, using a variety of adaptive, multi-step, and predictor-corrector methods. Since this computation can be time-consuming, all of these routines are implemented in C and can be run as threads spawned from the user interface. These functions place the computed point coordinates into a chunk which is available to the rendering thread for the display of interim results.

6.2 The streamline object

A new class definition is needed to manage calls to the calculation primitives. This new `<streamline>` class can inherit much of its behavior from the already-existing `<path>` class. Every instance of `<path>` stores a bundle of one-dimensional fields. It supplies methods for computing the bounding boxes of its fields and for rendering that curve using GL or PostScript.

The declaration of the new class declares it to be a subclass of `<path>`, to which it adds three new instance variables: *domain* , *vec-name* , and *my-thread* .

```
(defclass streamline
   (super path)
   (instance domain
             vec-name
             my-thread))
```

When any instance is created, it immediately receives the **isnew** initialization message. The corresponding method for our new class includes a call to the initializer of the superclass, and then stores the argument values into the internal state of this new instance. These three arguments are the bundle of the enclosing (usually 3D) domain, the name of the vector field (*e.g.* velocity, temperature-gradient) in which the streamline is to be computed, and the initial seed point from which the streamline to begin.

```
(defmethod streamline (ISNEW dom fld seed)
   (send-super 'isnew)
   (set!  domain   dom)
   (set!  vec-name fld)
   (self 'compute seed)
   self)
```

The sequence of points along the curve is computed by calling the compiled primitive function **streamline:calc**. The **compute** method provides a convenient wrapper for this routine. The primitive receives the 1D position field result buffer, the position and vector fields of the enclosing 3D bundle, and the initial seed point $\vec{x}_0$ for the integrated curve.

```
(defmethod streamline (COMPUTE seed)
    ;; Reinitialize this streamline,
    ;;    and allocate the destination buffer.
   (when my-thread (my-thread 'kill))
   (path 'reset)
(path 'alloc 'position 1000)
 ;; Do computation as a separate task.
(set!  my-thread
     (thread 'new
             streamline:calc
             (path   'export 'position)
             (domain 'export 'position)
             (domain 'export  vec-name)
             seed)))
```

The method does not call the primitive directly, but instead passes it and the arguments to the `<thread>` class, which spawns a task and returns an instance which serves as a handle for this computational task.

6.3 Using streamlines

Since the streamline class has been incorporated into the class hierarchy of SuperGlue, it is now available for use in any future application. This offers many benefits over the frequent alternative in which a new feature is accessible only as an option on a menu. Three examples presented here use instances of `<streamline>`, but specify the initial seed points of the curves in a different way.

The data used in these examples was computed by Ekaterinaris and Schiff [6]. It represents a "vortex breakdown" over a delta wing, moving at Mach 0.3 at 40 degrees angle of attack and a Reynolds number of 10^6. The grid contains slightly over 200,000 data points.

manual placement Using the standard 3D cursor instance (shown in figure 2), we can place a number of seed points in the breakdown region of the vortex. Exploration of this region of the data is greatly aided by direct manipulation and interactive response.

scripted placement Interactive placement is convenient for the exploration of some flow features, but placement at exactly specified points in the flow is often desired. A trivial Scheme expression was used in figure 3 to place a sequence of seed points along the leading edge of the wing.

computed placement Finally, a composition of functions can be used to automate the placement of streamlines. A 2D slice was extracted from the flow data. All grid points on this slice with a fluid density below some threshold were then used as seed points for a third set of streamlines, shown in a side view in figure 4.

The three sets of streamlines can be combined in a single image to produce a depiction of the vortex. This is shown in a side view in figure 5, and from upstream and above the wing in figure 6. The distinct placement methods complement each other and together provide a more useful tool for examining this data.

7 Conclusions

Our experience with developing and using SuperGlue has convinced us that our approach is sound. Development of new tools is rapid, fun, and endowed with high probability of a successful result. Problems with the system have appeared, but these are more irksome than serious.

7.1 Applications

SuperGlue has been used to create an animation of a proposed space station design (Globus [7]).

It has been used as an exploratory environment for the development of new (presently unpublished) visualization techniques.

It was used by one of the authors as the development platform for an improved method for the construction of stream surfaces [8].

It currently lacks some of the features required for unassisted use by scientists, but we are adding this "user-friendliness" as quickly as we can. We believe that having invested a great deal of time in laying the foundation, we shall be able to construct the rest of the house much more quickly.

7.2 Problems

We did not fully appreciate the sheer magnitude of code required in an interactive visualization environment. SuperGlue is now about 35000 lines of code, split about equally between Scheme and C. After applying three man-years of effort to extending a previously existing interpreter, we have only made a dent in our list of desired features.

The difference in the speeds of interpreted Scheme and compiled C can be as much as one hundred-fold. For many portions of the system, this difference in speed is not important; in others, it is unacceptable. An optimizing native code compiler for Scheme and a faster garbage collector would allow Scheme code to be used for a greater proportion of the system. Commercial LISP implementations provide this speed, but we needed a system which could be fully modified for our particular goals.

We did not fully anticipate the great resistance we have encountered regarding the use of Scheme. Dialects of LISP are still regarded with suspicion or disdain, perhaps in supercomputer centers more than anywhere else.

7.3 Future plans

The SuperGlue project is open-ended by design; progress is made by growing the class hierarchy. Certain features hold particular interest for us and we expect to spend the coming year implementing these items:

publication tools Interactive color raster displays are a valuable exploratory tool, but publishable images are a necessity. The realities of modern publishing still require line art for the majority of archival publications.

virtual reality Levit and Bryson [3] have shown that the visualization of complex flow fields can be greatly aided by the technology of virtual reality. In a virtual reality environment the display must be repainted in real-time, so all of the drawing code must be migrated into C where it will be immune from intermittent pauses caused by garbage collection.

unsteady data The state of the art in computational fluid dynamics has reached the stage of simulating 3D unsteady flows. The massive data storage requirements almost certainly demand a distributed implementation using workstations connected to supercomputers over high-speed networks.

7.4 Summary

No tool is right for every job; no language is right for every algorithm. The bilingual structure of SuperGlue provides a flexible platform from which to attack the varied requirements of interactive visualization, from the tuning of a user interface to the processing of a large dataset.

Chunks and memory-mapped files are compact and efficient mechanisms that allow data to be managed and examined from the Scheme command layer, yet rapidly processed by compiled numerical and graphical routines. Bundles simplify manipulation of CFD datasets. Destructors are an efficient mechanism that allow external resources to be returned when they are no longer needed.

Programming with an interpreter allows convenient exploration of alternative implementations. The class hierarchy provides a clean mechanism for the reuse of code. Together, these features help reduce the time required for the implementation of new features. The programming members of a visualization team are able to respond in a timely manner to the needs of their scientist-clients.

Acknowledgements

We wish to thank Tom Lasinski, who has steadfastly supported our efforts. We thank Al Globus, Kris Miceli, and Michael Gerald-Yamasaki, who have tested early versions of the code. JH wishes to thank Mary Hultquist for her encouragement and tolerance. We also thank David Betz for his fine implementation of Xscheme, and John Ekaterinaris and Stuart Rogers for allowing the use of their flow data.

References

[1] Gordon V. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–27, San Francisco, CA, October 1990.

[2] Brian Beckman. A scheme for little languages in interactive graphics. *Software-Practice and Experience*, 21(2):187–207, February 1991.

[3] Steve Bryson and Creon Levit. The virtual windtunnel: An environment for the exploration of three-dimensional unsteady fluid flows. In *Proceedings of IEEE Visualization '91*, San Diego, CA, 1991. To appear in Computer Graphics and Applications, July 1992 ???

[4] Pieter G. Buning and J.L. Steger. Graphics and flow visualization in CFD. In *AIAA 7th CFD Conference*, pages 162–170, Cincinnati, OH, July 1985. AIAA Paper 85-1507-CP.

[5] D.M. Butler and M.H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3:45–51, Sep/Oct 1989.

[6] J.A. Ekaterinaris and L.B. Schiff. Vortical flows over delta wings and numerical prediction of vortex breakdown. In *AIAA Aerospace Sciences Conference*, Reno, NV, January 1990. AIAA Paper 90-0102.

[7] Al Globus. The design and visualization of a space biosphere. In *10th Biennial Space Studies Institute / Princeton University Conference on Space Manufacturing*, Princeton, NJ, May 1991.

[8] J.P.M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of Visualization '92*, pages 171–178, Boston, MA, October 1992.

[9] Silicon Graphics Inc. IRIS Explorer user's guide, January 1992. Document Number 007-1371-010.

[10] Thomas C. Palmer. A language for molecular visualization. *IEEE Computer Graphics and Applications*, 12(3):23–32, May 1992.

[11] Jonathan A. Rees and William Clinger. Revised^3 report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12), December 1986.

[12] S. Rogers, D. Kwak, and U. Kaul. A numerical study of three-dimensional incompressible flow around multiple posts. In *AIAA Aerospace Sciences Conference*, Reno, NV, January 1986. AIAA Paper 86-0353.

[13] Craig Upson et al. The application visualization system (AVS): A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[14] Joel Welling, Chris Nuuja, and Phil Andrews. P3D: A lisp-based format for representing general 3d models. In *Proceedings of Supercomputing '90*, pages 766–774, Jan 1990.

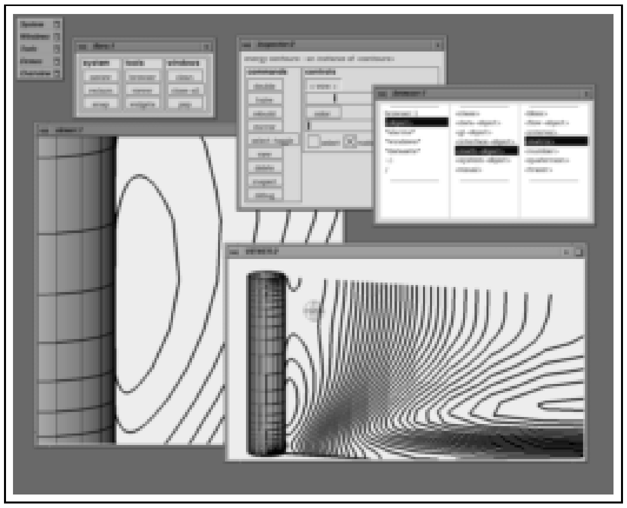
Figure 1: Application interface of Superglue, showing two scenes, an inspector, and a hierarchy browser.

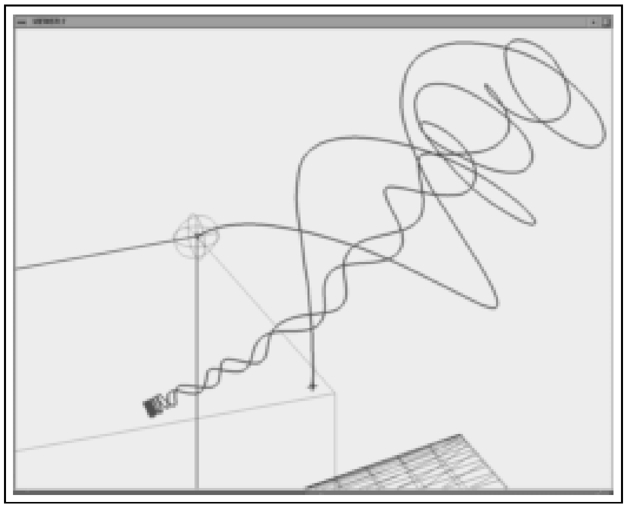
Figure 2: Close-up view of the mouse-directed placement of seed points in the breakdown region.

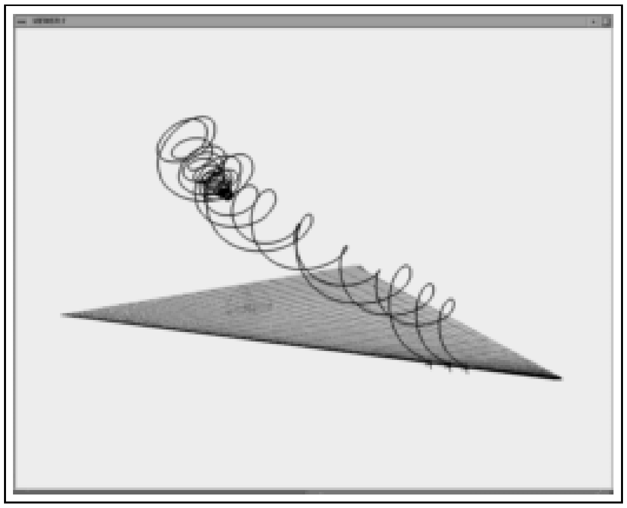

Figure 3: Scripted placement of seed points along the leading edge of a delta wing.

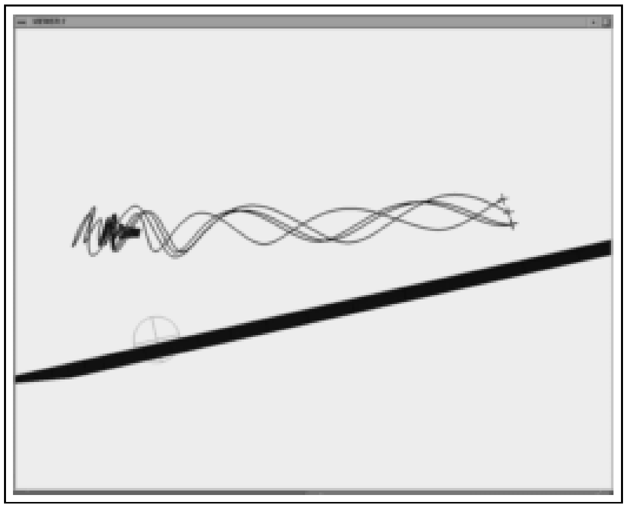

Figure 4: Side view of the computed placement of seed points in the low-density vortex core.

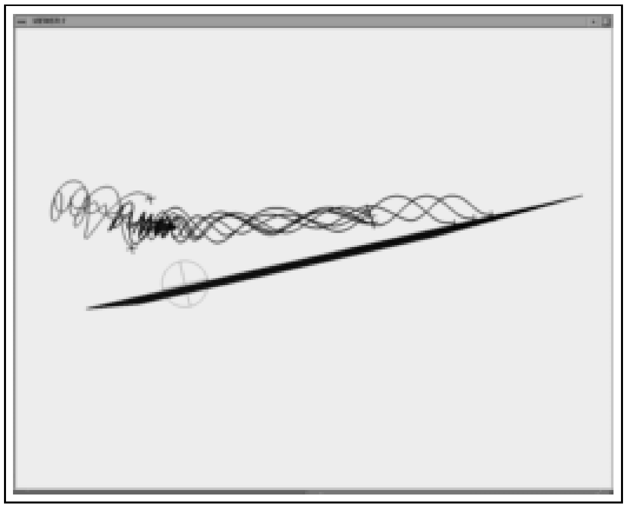

Figure 5: Side view of the completed visualization of the vortex breakdown over a delta wing.

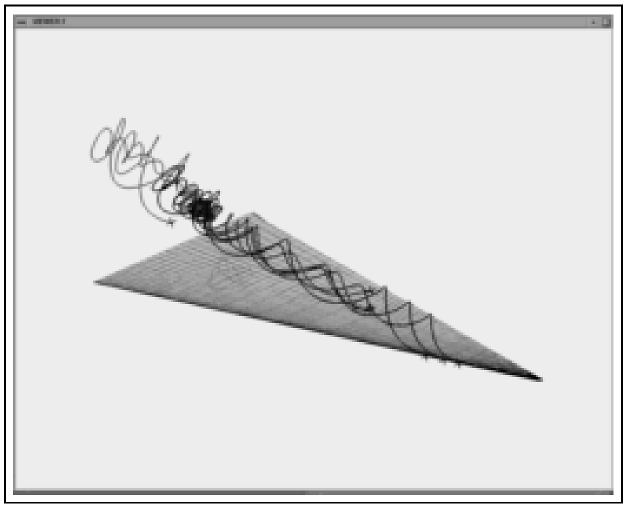

Figure 6: Three-quarter view of the vortex breakdown over a delta wing.